# Testing

## Group 18

### Team B

Olivia Betts
Zac Bhumgara
Nursyarmila Ahmad Shukri
Cameron Duncan-Johal
Muaz Waqas
Oliver Northwood
Teddy Seddon

A) Over the course of our project, we used a combination of automated unit tests and manual testing. Generally, automated testing was more made use of for the classes for food and actors, whereas manual testing was used for classes for the different screens in the game and what they would render. We used a mixture of whitebox and blackbox testing, but the majority of our blackbox tests were manual. Bearing the testing pyramid in mind, we focused mostly on unit tests - these were also the easiest to implement.

The first step we took was assigning roles to everyone in the group. In terms of testing, Muaz and Teddy were assigned to make and run the tests, as well as complete the testing report. We decided to use Junit 4 for our automated testing. Junit makes it easy for the machine to recognise the presence of a test due to the @Test keyword, as well as allowing us to use the *assert* keyword to check for preconditions.

The first stage was to create an environment in which we could start testing the original game. Therefore a GdxTestRunner class was made, which allowed us to run the tests using the Headless configuration. This was useful, because it meant that there was no actual graphics of the game being produced, and it would be easier to just test the logic.

Upon adding the correct dependencies, the first tests which we made were asset tests. These would check for the presence of assets essential to the running of the game in the assets folder. Therefore, they were important for the overall usability of the project. We made asset tests for the assets related to recipes and customers, as well as the music.

After the asset tests were complete, we moved onto logic tests. We started with testing the individual classes for correctness and usability e.g. Burger class, Customer class. This allowed for easy detection and correction of issues at class level. We would test all the methods in the class and make sure that they were all working as expected. We also put in some incorrect cases and made sure that they failed and the game had a suitable way to counter this i.e. it didn't just crash.

After that, we planned to move onto testing some of the user input elements. Unfortunately, at this point, we came across a problem and that was that most of the code was in the GameScreen class, which had not made any use of lazy evaluation. This meant that the code was not very testable. Every time the GameScreen class was called, the spriteBatch was called alongside it (but spriteBatch could not be null) - for this reason, we struggled to run the Junit automated tests in Headless configuration. Our solution for this was to test most of the GameScreen class using manual testing. We did end up making some tests for the GameScreen class, but they did not run as expected. For this we also implemented Mockito. We were unable to automatedly test most of the UI elements.

We then moved onto manual testing. Using the requirements, we created tests for all of the requirements, as it is good practice to have tests for each requirement. For all the tests we created success and failure scenarios and tested them all, documenting thoroughly in a Word document, and adding pictures to explain the current state of the game.

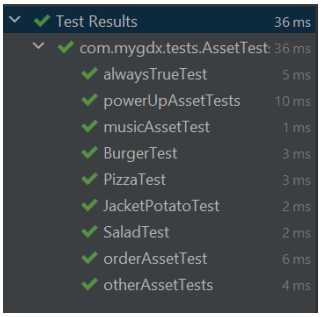
Throughout the development of the game, we continued testing our added code to make sure that we had not broken previous functionality with the new additions.

At the end of the testing, to make sure that we had covered every requirement, we created a traceability matrix, as well as a testing report. Both of these documents were useful in helping us to figure out the overall performance of our project for our tests.

In terms of the overall timing of the tests, we carried out most of the asset tests at the start of the testing cycle. This was to ensure that all necessary assets were present before proceeding with logic tests. The logic tests were then conducted during the development process, to double check if the new implementations are correct. The manual testing was done at the end of the development process, once the game was approaching completion.

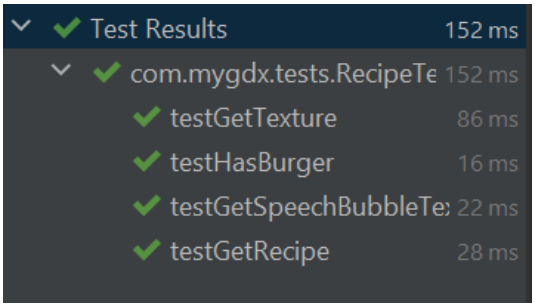
B) Firstly, we made a traceability matrix, to link our tests to our requirements. This meant that it was much easier for us to keep track of what requirements we have already tested, and which tests this is contained within. We found this to be a more representative indicator of what and how much we needed to test as compared to code coverage. One the traceability matrix, the failed or incomplete tests were starred. The full traceability matrix can be viewed on our website (URL provided on next page), but an excerpt of the traceability matrix can be seen below:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2.1 | x | | | | | | | |
| 2.2 | | x | | | | | | |
| 2.3 | | | x | x | | | | |
| 2.4 | | | | | x | | | |
| 2.5 | | | | | | x | x | |
| 2.6 | | | | | | | | x |
| 2.7 | | | | | | | | x |

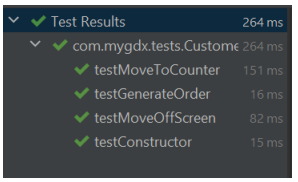


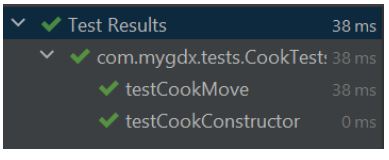

The first tests we implemented were asset tests. For these, we checked for the presence of the assets essential for the game to run. This mostly consisted of items of food. We had 9 main asset tests, which checked for all the .png images for the food and the .mp3 file for the music in the background of the game. 100% of the assets tests passed, but as they are not actually testing the code, we had 0% line coverage with them.

We then moved onto logic testing the food classes. Each of the four food classes (salad, burger, jacket potato and pizza) all had 4 methods in them - 3 getter methods, for the recipe, the texture and the speech bubble texture. The 4th method would check whether the necessary ingredients were present to make the item. In our automated unit tests, we checked the getter methods to see if they were actually returning the attribute required. Our last test was to check the correctness of the class and its corresponding logic. For example, we checked to see if the ingredients for a burger (patty, lettuce and buns) actually combine to make a burger. We then checked some other cases too, which included inputting the ingredients into the stack into a different order, adding wrong ingredients and adding duplicate ingredients. 100% of tests for all four of these classes passed and we achieved 46% line coverage.

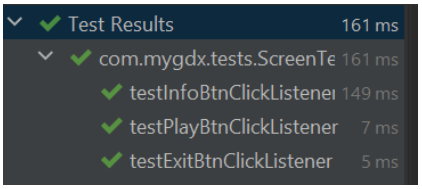



The logic testing with the cook and customer was a bit more difficult due to the fact that a lot of their methods were written in the GameScreen class. The methods in the cook and customer class were tested successfully, with 95% and 100% line coverage respectively. All tests for the customer class passed, however, for the cook class the test for flipping the patty did not work with our dependencies. For this reason, we have commented it. The cook and customer movement were tested successfully, as well as the customer's ability to generate an order, and move offscreen.




We tested for the difference in buttons if you hover upon them with a cursor in all of the screen classes. However, we were unable to test most of the methods in the screens classes due to the difficulties in testing UI. Unfortunately most of the game code was in the GameScreen class, so instead to



make up for our lack of automated testing, we performed manual tests to fulfil the rest of the requirements.

We have a few non-working tests in our screen test classes. These do not work due to the fact that we have not used lazy evaluation whilst writing our code. Therefore, everytime GameScreen was called, it would ask for a SpriteBatch. However, as we were running the game in Headless configuration, the spriteBatch would be null. This would give an error and it would not allow the test to run. Due to time constraints, we were unable to refactor the code sufficiently to combat this. A couple of other tests in the SettingsScreenClass were also not completed; this was also due to time limitations.

As you can see from our report, we achieved 30% class coverage overall. The full coverage report is on our website and the URL has been provided on the next page. Obviously, this was a low percentage, therefore we carried out a large number of manual tests to properly test our game. These have been fully documented on our website, including the steps we took, the expected and the actual outcomes. This can be found on our website and the URL is linked in the next question. In this way, we managed to finish off our traceability matrix, by linking every requirement to a test.

Overall Coverage Summary

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| all classes | 30% (21/70) | 21.4% (63/294) | 18.2% (332/1820) |

Coverage Breakdown

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| com.mygdx.game | 50% (3/6) | 38.1% (8/21) | 43% (58/135) |
| com.mygdx.game.Food | 100% (6/6) | 34.4% (11/32) | 46.8% (73/156) |
| com.mygdx.game.Screens | 6.8% (3/44) | 3.6% (6/165) | 1.4% (16/1121) |
| com.mygdx.tests | 80% (4/5) | 54.8% (17/31) | 63.7% (107/168) |
| com.mygdx.tests.RecipeTests | 0% (0/4) | 0% (0/20) | 0% (0/88) |
| com.mygdx.tests.ScreenTests | 100% (5/5) | 84% (21/25) | 51.3% (78/152) |

For our manual tests, we followed a structural layout in terms of documenting the tests. Every test (like the automated tests was given a unique code and name), so they could be easily referenced in the traceability matrix. This was followed by a short description and the requirement(s) which the test relates to. Following this, we had 4 rows, for steps taken in the test, expected outcome, actual outcome and whether the test passed or failed. We then had a row for failure cases, edge cases and other notes, just to explain what other cases were tested for, which could not be inferred from the steps. Finally a final row for some pictures to show proof of testing. Below is an example of one of the manual tests and the corresponding documentation. The full documentation can be found on our website.

| | |
|---|---|
| Identifier | TEST_UR_SWITCHING_COOKS (2.1) |
| Short Description | This test will check whether we are able to switch cooks with the number keys 1,2,3. |
| Related Requirement(s) | UR_SWITCHING_COOKS |
| Author | Muaz |
| Steps | After starting the game, click on 2 to switch to second cook. Then click on 3 to switch to 3rd cook. |

| | |
|---|---|
| | Then click back on 1 to switch back to the first cook. |
| Expected Outcome | The cooks should switch based on the arrow key clicked. |
| Actual Outcome | The cooks switched based on the arrow key clicked. You can tell which cook is selected due to the selected arrow on top of the cook. Also at the bottom right it shows you which cook is selected. |
| Failure/Edge Cases | Clicked on other keys to see if cooks would change - they do not<br>Clicked on cooks to see if that would change them - they do not |
| Status (Pass/Fail) | PASS |
| Pictures |   <br>Picture before and after clicking on key for number 3. |
| Notes | We tried these in both endless mode and scenario mode and for all difficulties, just to make sure that our test results were reproducible across all modes. |

It is important to stress that our testing continued throughout the development of our project. Every time someone in the implementation team would add a new functionality, we would run our tests to make sure that the previous code was not affected was not affected by the new commits. Running our tests was very intuitive and it meant that after the tests were made then anyone could run them, without much prior knowledge. At the end of our implementation period, we ran the automated tests again, and carried out all the manual tests and analysed the results. The analysis of our results showed that in general the testability of our game is not great. Most of the game logic is contained within UI classes which are untestable. Therefore our tests were not fully complete, but every effort was made to make them as complete as we could with the previous group's coding. However, our manual tests proved that most requirements of the game were met to a high standard.

UR_USER_EXPERIENCE, UR_LICENCE, UR_READABLE_CODE and UR_FAMILY_FRIENDLY_CONTENT are all requirements which cannot be manually tested definitively per say. The opinion on all these is subjective, so instead we have made maximum efforts to try and achieve all of those requirements as explained by the client, throughout the production of our product. Nevertheless, we have still linked some tests to these requirements, to show how our game is easy to use. This can be found in the manual testing documentation.

We ran some integration tests through the use of continuous integration. However, this is explained in more detail in the continuous integration section.

In terms of future maintenance, it would be beneficial to refactor the code and move some of the code from the UI classes into individual object classes. This would make the overall coverage of the code better as well. However, overall our testing was a success due to the fact that we were successfully able to test for all our requirements and the tests proved that we achieved the majority of them.

c)
Documentation of Manual Testing:
https://teambeng1.github.io/TeamBPart2.github.io/pdf/Documentation%20of%20Manual%20Testing.pdf

Traceability Matrix:
https://teambeng1.github.io/TeamBPart2.github.io/pdf/TraceabilityMatrix.pdf

Coverage Report:
https://teambeng1.github.io/TeamBPart2.github.io/htmlReportTesting/htmlReportTesting/index.html
https://teambeng1.github.io/TeamBPart2.github.io/img/coveragereport.png

# Test reports

# Testing material URLs

Software Testing Report [22 marks]:
a) Briefly summarise your testing method(s) and approach(es), explaining why these are appropriate for the project. (5 marks,  1 page)
b) Give a brief report on the actual tests, including statistics of what tests were run and what results were achieved, with a clear statement of any tests that are failed by the current implementation. If some tests failed, explain why these do not or cannot be passed and comment on what is needed to enable all tests to be passed. If no tests failed, comment on the completeness and correctness of your tests instead (12 marks,  3 pages).
c) Provide the precise URLs for the testing material on the website: this material should comprise the testing results and coverage report generated by your automated testing tooling, and descriptions of manual test-cases that you designed to test the parts of the code that could not be covered by your automated tests (5 marks).

It is good testing practice to have tests created for all of the requirements for the project. Therefore as our first step for the testing, we looked through the requirements and brainstormed what tests could be relevant for each.